BioSim User Guide

Marc Blenkiron
Marc@Blenkiron.com
Institute for Computer System Architecture
School of Informatics
Edinburgh University

Version 1.07c

June 2004

Contents

1 Introduction **3**
- 1.1 Purpose . . . 3
- 1.2 Graph . . . 3
  - 1.2.1 Definition . . . 3
  - 1.2.2 Fluid Mapping . . . 3
  - 1.2.3 Edges . . . 3
  - 1.2.4 Multiple Media . . . 4
- 1.3 Nodes . . . 4
  - 1.3.1 Level of Abstraction . . . 4
  - 1.3.2 Modules . . . 4
  - 1.3.3 Granularity . . . 5
- 1.4 Technical Note . . . 5
- 1.5 Search Capacity . . . 5
- 1.6 Making an Experiment . . . 5
- 1.7 Other Considerations . . . 6

2 User Guide **7**
- 2.1 Stages . . . 7
  - 2.1.1 Specifying Modules . . . 7
  - 2.1.2 Connection Model . . . 7
  - 2.1.3 Packet Types . . . 7
  - 2.1.4 Plug-in Types . . . 7
  - 2.1.5 Compiling a BʔSim program . . . 8
  - 2.1.6 Tailoring a script file . . . 9
  - 2.1.7 Programs for Nodes . . . 9
  - 2.1.8 Running a BʔSim program . . . 10
  - 2.1.9 Exporting Settings . . . 10
  - 2.1.10 Exception Mechanism . . . 10
  - 2.1.11 Plug-in Database . . . 10
  - 2.1.12 External Channels . . . 10
- 2.2 Specifying Modules . . . 11
  - 2.2.1 Format . . . 11
  - 2.2.2 Parameters (MODULE) . . . 11
  - 2.2.3 Storage . . . 11
  - 2.2.4 Instructions (INTERFACE and INSTRUCT) . . . 12
  - 2.2.5 Input, Output and Actuation . . . 12
  - 2.2.6 Commands . . . 12
  - 2.2.7 Command Parameters . . . 13
- 2.3 Connection Model . . . 13
  - 2.3.1 Note . . . 13

3 Examples **14**
- 3.1 Introduction . . . 14
- 3.2 Preparation . . . 14
- 3.3 Examples . . . 15
  - 3.3.1 Processor . . . 15
  - 3.3.2 Leader Election . . . 19
  - 3.3.3 Aggregating Amoebae . . . 27

A Programming Interfaces **46**
- A.1 Connectivity (Engine) . . . 46

1 Introduction

1.1 Purpose

BiSim is a discrete time, event driven simulation platform, the purpose of which is to explore models of peer-to-peer networks composed of interacting devices, from asynchronously operating scatterings of silicon specks to biological cells. Runs are performed in a computational context to analyse their behaviour, performance and to allow the potential for *automated* model optimisation.

Each device is considered to be composed of a number of connected functional units, which may themselves be composed of other such units, and specified in an interpreted script file. Provisions have been made to support complex, time-changing models.

Its support for modularity and flexibility allows devices to have multiple, time and event varying communication channels, such as radio and infra-red or inter-cellular and intra-cellular, with the ability to express characteristics such as external effects on signal strength or chemical diffusion characteristics and allow an end-user (or an automated tool) to configure and use these in a simple script file.

Such emphasis on modularity, from virtual machine register-level operations, to user-defined and communication behaviour is intended to provide one approach to help complex models to be more easily understandable, modifiable, configurable, usable and to potentially support time-saving techniques such as automated runs and analysis.

One prime challenge of such networks is to discover how it may be possible to organise such a vast collection of simple components to cooperatively perform a task.

1.2 Graph

1.2.1 Definition

In the Computer Science discipline, a network, or *graph*, typically refers to a set of components, or nodes, some of which are connected by links, or *edge*s. These nodes are able to communicate, or pass information, directly with those nodes they are linked to. Examples include computers linked with cables, cells linked by chemical diffusion cellular pathways inside a cell.

Computer Science has evolved tools that are able to manipulate and analyse such graphs and this is how BiSim internally represents the way in which the entities it is simulating are connected.

1.2.2 Fluid Mapping

Some models of interacting devices tie the way in which nodes are connected to a particular model of spatial locality, forces, or movement. BiSim takes a more general approach and allows the user to provide, or plug in a mapping from their own user-defined connectivity model to an underlying graph. This mapping can also be fluid, or dynamic - that is while an experiment is running the graph itself can be changing. BiSim will automatically update the graph's connections as the connectivity model changes according to the user's rules.

A simple, grid-based, two-dimensional model is provided, on which a user may add their own mapping rules.

1.2.3 Edges

BiSim associates distance and signal strength information with each edge of a graph. Distance determines how long a message, or molecule may take to travel from one node to another. For example, in the context of cells, if a chemical is diffused at a corner of a densely packed collection of cells, those nearer to the corner will be more likely to receive a higher concentration of it initially.

BiSim does not attempt to model all possible scenarios for this type of message passing. Instead a more general approach has been adopted that allows a user to plug in their own model of message broadcasting, based on the

connections of the node that is sending a message. It is also possible to specify what happens when messages collide, that is are sent at around the same time, or arrive at a destination at the same time.

Plug-ins for a crude approximation of chemical diffusion and lossless computer packet passing are included.

1.2.4 Multiple Media

Sometimes one graph may not be enough, or not convenient. It may be necessary, for example, to model inter and intra cellular communication on two different graphs that have differing rules for diffusion depending on whether molecules are inside or outside a cell membrane.

BiSim is able to support multiple graphs - a connectivity model may provide a mapping to more than one graph and each graph may have its own plug-in rules.

1.3 Nodes

1.3.1 Level of Abstraction

Depending on the desired level of abstraction, a node could, for example, be a single cell, a collection of cells, or one component that makes up a cell.

Different types of node can be specified, such as one for each type of component a cell comprises. It is also possible to have nodes with different levels of abstraction in the same experiment. This can be useful if one wishes, for example, to simulate a large experiment of nodes but only wants to see more detailed behaviour in a small, chosen clump of nodes.

1.3.2 Modules

The user may specify their own types of node with which to experiment. To aid the possibility of simulations using more complex node definitions, an object-oriented approach has been adopted, that is, the user may define a number of modules where each module provides some functionality, and a node's composition, or architecture, is specified by those modules which it uses hierarchically. For example, if a type of node is a cell, its modules may correspond to the components that make up a cell. This also allows an experimenter to more easily try out a number of different types of node, as well as allowing a toolkit of components that can be plugged together to build new types of node without having to recompile the source code every time.

Each module may provide functionality for the following:

- Storage : A place to store information, for example, a program, chemical concentrations or behavioural rules.
- Output : How to determine which messages are broadcast, such as which molecules pass through a semi-permeable cell membrane.
- Input : What occurs when a message is received.
- Actuator : The node can send messages to a connectivity model here. The connectivity model may respond by, for example, changing a graph.
- Instructions : A list of things that this cell is able to compute. For example, this may be a list of chemical reactions that can occur inside a cell, or could be the instruction set of a computer processor. A cell's program may use these instructions.

A node's input, output and actuator processing happen independently of its internal behaviour, or program that utilises its instructions. Each of these types of action can have an associated time and power cost that it takes them to complete. A program for a type of node can be written using the instructions of any of the modules from which it is composed, in other words, a node may be programmed to perform any of the abilities it has been given. Such a program may be loaded separately from an external assembly file. It is also possible to make modules composed of other modules.

A compiled script allows the user to describe a module fully, optionally utilising the C/C++ programming language where required.

1.3.3 Granularity

One aim of BiSim is to be able to analyse macro-events at micro-levels as small as the granularity of the instructions as specified by the user in their modules. In other words, if some interesting or unexpected macro behaviour occurs, we may wish to discover the sequence of smaller events that lead to it. For this reason, BiSim is able to output these events in the form that a user specified in instructions.

1.4 Technical Note

From an implementation perspective, we are not just allowing the user to use an object-oriented programming language to specify models, as could be done simply, for example, in C++, Java or ADA, we are also compiling these modules on-the-fly into a number of object-oriented virtual machine types that are able to compile programs written their respective vocabularies, on-the-fly and interpret them with meta-information for spying on their computations. This is all done automatically by BiSim and the user does not need to know about the intricacies of this mechanism.

That is, the architectures are soft and this allows considerable power of configurability and potentially simplifies the end user interface, both described below.

1.5 Search Capacity

Given some evaluation criteria, such as power consumption minimisation, it is possible to perform a directed search over node compositions and other settings in an automated manner by using an external script, say in perl, to read an output indicator from BiSim and then in turn manipulate these compositions and settings.

1.6 Making an Experiment

To create an experiment, the user specifies; the available functional unit modules and their parameters; the types of node composed of these modules; an optional, single connectivity model; available communication graph rules and the nodes in the experiment along with which programs they use.

Pre-made types, rules and programs supplied by BiSim can also be substituted for these or augmented by the user.

Many parts of the experiment, such as the composition of nodes can be described in an interpreted script, allowing an end-user unfamiliar with C/C++ (or a search tool) to utilise the platform.

1.7 Other Considerations

Other aspects that have been borne in mind while BiSim was being designed:

- A flexible interface for the specification of experiments. A simple 2D graphical front end is provided but there is plenty of room to extend this to take advantage of available simulation data.
- A defensively designed, robust platform, as the user is able to create their own experiment executable attempts have been made to prevent a mistake going unnoticed through all parts of the platform.
- Although Linux is the first target for testing platform releases, the code is intended to compile on Windows and Mac OS X and has been verified recently by users with preferences for these operating systems.
- Nodes have the capacity to be reprogrammed while the simulation is running. It is also be possible to modify a node's architecture.

2 User Guide

2.1 Stages

The steps required to create an experiment form a number of stages. Some of these stages may be omitted by using components that other people have made or defaults supplied with BiSim. An end user that wishes to try experiments with a pre-made model, for example, would only require an overview of the final stage of those listed below.

2.1.1 Specifying Modules

A number of functional modules may be specified using a scripted syntax. These module descriptions are compiled and added into a palette of options inside a BiSim program, from which nodes may be later composed. It is possible to make use of C/C++ within the script syntax, however it is also possible to avoid the use of much programming knowledge, for more simple operations. Each module specification may declare that it can be configured using a number of parameters, such as numbers, labels, and even other modules. Provided modules include a subset of a *Mips R3000* instruction set, a broadcasting modem and a simple module able to represent molecules passing in and out of a cell.

2.1.2 Connection Model

This section shows how the possible networks of connections between nodes may start out and change over time, or in response to events, such as movement. It is not scripted and if pre-made models are not being used it requires the use of C++ to describe a model based on a starting point described in section *Programming Interfaces*, or more easily, augmenting an existing model, such as the *Grid2D* two-dimensional space model.

2.1.3 Packet Types

A packet type states the information contained within a message that is passed between nodes over a connection graph. Packet types are described in a script file. Examples range from plain binary data blocks to chemical concentrations.

2.1.4 Plug-in Types

The purpose of a plug-in is to allow a user freedom both to describe a set of functions that are used by the BiSim engine and also to be able to select which one to use by name (and its settings) in an interpreted script file.

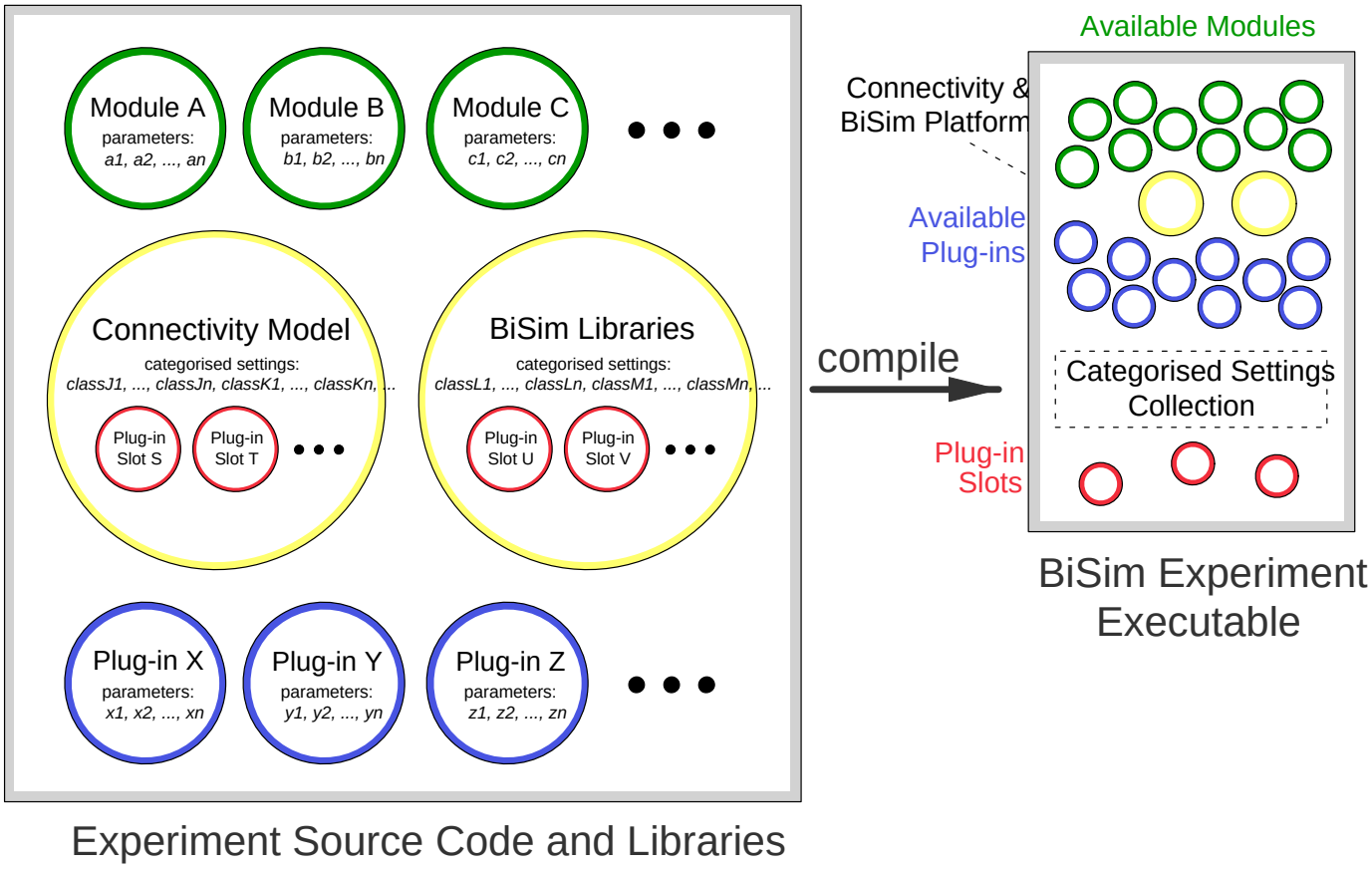
For example, when a packet is broadcast over a connection graph, if it is an email, it may arrive as a perfect copy of the original on each computer it is sent to. If it is a group of molecules escaping from a cell membrane a number of nearby cells will not receive copies, but each may be seen to receive a proportion, the quantity of which is calculated.

Plug-ins are used where there are so many possibilities for differing experiment types that users may want to perform, that catering explicitly for all of them is not feasible. Instead, a generic mechanism is provided, along with a number of common examples from which users may design their own, or modify existing ones.

2.1.5 Compiling a BiSim program

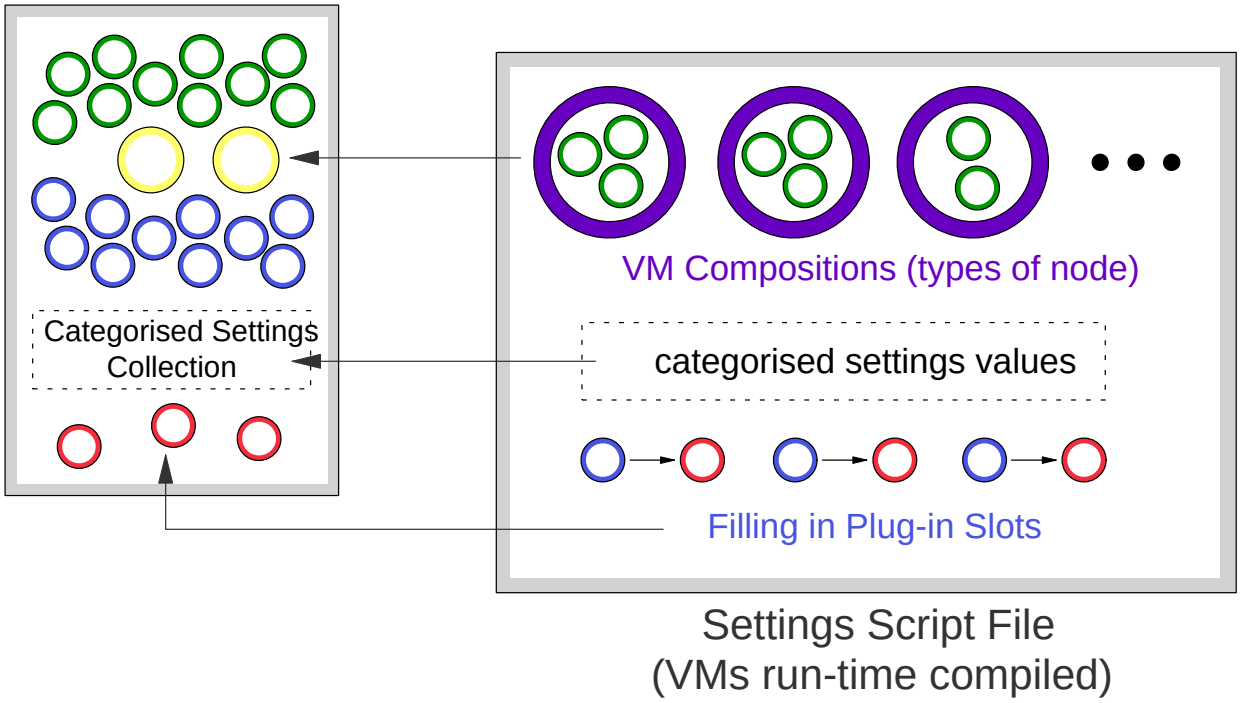
Once the previous stages have been accounted for, a stand-alone program may be created with which experiments can be performed. A set of modules, a set of plug-ins and a connectivity model are compiled with BiSim's libraries to produce a binary executable.

The modules and plug-ins may have optional parameters so that they can be configured after compilation. The connectivity model and BiSim's libraries can provide slots into which plug-ins can be added to allow the user to modify the simulation platform's behaviour. These two components can also be parameterised by settings that are available to the user to customise an experiment.



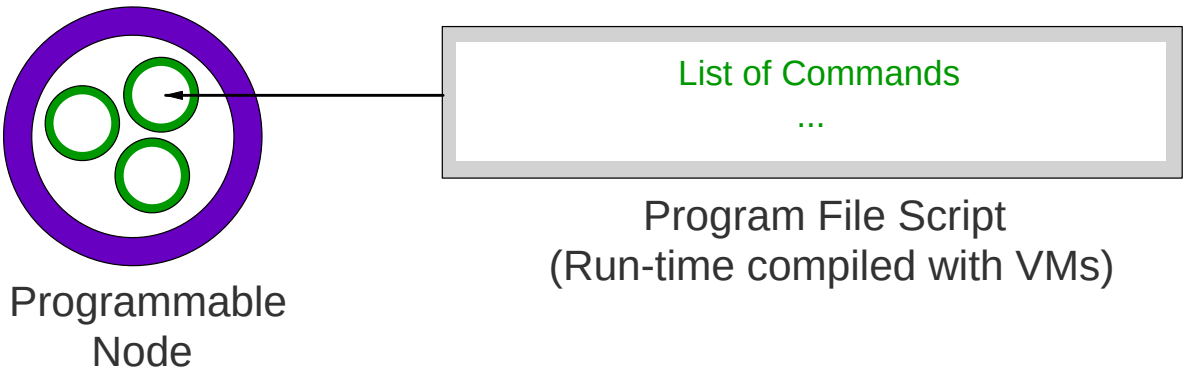
2.1.6 Tailoring a script file

Once an experiment has been compiled into a BiSim program it can be run by providing it with a settings script file. This file contains settings that may be modified by the user. It may also contain node specification (virtual machine compositions), made up of modules and their parameters. Different kinds of experiment will have different required settings. The experiment script file is read at run-time, that is, it is interpreted.



2.1.7 Programs for Nodes

A program for a node may utilise any of its modules' instructions that the user has defined. Recently, the ability has been added to load a program from an external text file in a generic assembly format, and to have it automatically compiled against a node's modules at run-time. It is also possible to hard-wire a program into a node through a macro interface if desired.



2.1.8 Running a BSim program

A BSim program is a command-line tool with a number of options to run experiments and interrogate its contents, for example, ask what modules it contains that can be used to make different types of node. The user's information for configuring and customising an experiment may be given to a program in the form of a settings script file.

2.1.9 Exporting Settings

When designing a new connection model, using C++, it may be desirable to invent a number of settings specific to this model that an end user can change in an experiment script file. This may be anything from the number of nodes used, to choosing a type of plug-in to use. Once settings are declared, they are added *automatically* into the script mechanism for interfacing with an interpreted script file.

2.1.10 Exception Mechanism

As a convenience, it is possible to use the C++ exception generating mechanism that is used in the BSim implementation. In doing so, BSim's built-in debugging and exception reporting will be utilised automatically if a problem is encountered.

2.1.11 Plug-in Database

When designing a new connection model, using C++, it is possible to generate a *category* of plug-in type, allowing others to make and use their own plug-ins based on this description.

2.1.12 External Channels

These channels manage how data is exported to an external tool in simulator real-time, such as BSim's simple graphics rendering engine. They can also be used for exporting of statistics.

2.2 Specifying Modules

This shows the way a module type can be described using a scripted format. The valid syntax of a module will firstly be shown, then a brief description of each section to set it in context with (*Modules Introduction*, 1.3.1) and (*Modules Overview*, 2.1.1). Finally a table will be presented summarising all of the possible commands and parameters. Please refer to (*Examples*, 3) for a range of different sample modules.

2.2.1 Format

A module takes the following format:

```
.  MODULE(size)
.
.     (
.          typename, [basename | BASE(packet)], (baseargs)
.          [, datatype, identifier, default]*
.     )
.
.     INTERFACE
.          [DECLARE(op, hook, arg0, arg1, arg2, time, power)]*
.     STORAGE
.          [LINK(identifier)]*
.          [DATA(dataname, datasize)]*
.     FUNCTIONS
.          [INSTRUCT(typename, hook, var0, var1, var2)
.               usercode [NOASSIGN | ASSIGN(value)]
.          ]*
.          [INPUT(packet, var)
.               usercode COST(time, power)
.          | ]
.          [OUTPUT(packet, var)
.               usercode [SEND(packet, var) | ] COST(cost)
.          | ]
.          [ACTUATOR(packet, var)
.               usercode [SEND(packet, var) | ]
.          | ]
.     [NOSPACE | SPACE dataname [, dataname]* ; ] ]
.
.  CLOSE(typename)
```

2.2.2 Parameters (MODULE)

When we declare a new type of module, we specify how many parameters it takes, and what those parameters are. These are the parameters that a user will later be filling in or adjusting in an interpretted settings file. BiSim is able to support strings, floating point numbers and integers as parameter types. It can also accept another module *type* as a parameter to create dynamic compound types. If a module is able to send or receive messages, the *type* of these messages is also specified here.

2.2.3 Storage

We declare what data this module needs to store in this section. If dynamic compound types are being used, this section can also link them into this module's context.

2.2.4 Instructions (INTERFACE and INSTRUCT)

Here we present a list of operations that this module is able to perform in the `INTERFACE` section. Each interface operation is tied to a function template that is declared in an `INTRUCT` section. Each `INTERFACE` operation can also specify that it takes up to three parameters and whether they will be immediate values or a value to be read or written to via a node's storage space.

Inside an `INSTRUCT` declaration, values for parameters that a user will later specify in a settings file can be accessed by prefixing the name of the parameter with, `self.` - for example - `self.threshold`. A node's storage can be accessed in a similar way - for example - a data area called, `clock`, can be accessed through, `NODE[self.clock]`. Decisions and data manipulations can be performed inside a function using C++. A function terminates by assigning a result to its operation.

2.2.5 Input, Output and Actuation

When a message is received from another node's module (via `INPUT`), a message handler can be implemented to specify what action occurs. Similarly, in response to some internal state of a module, messages can be sent to other nodes (via `OUTPUT`) or action messages to a connectivity model (via `ACTUATOR`). Parameters and data members can be accessed in a similar manner to instructions, except that the `self` prefix is not required here.

2.2.6 Commands

MODULE	begins a module specification
BASE	creates a base class module type using a given packet type
INTERFACE	starts a list of externally visible instructions
DECLARE	adds an instruction type into the module's vocabulary
STORAGE	begins the list of storage locations
LINK	links child modules given as parameters, into this module
DATA	specifies a storage location
FUNCTIONS	begins the section where blocks of code may be defined
INSTRUCT	declares a block of code used as an instruction template
NOASSIGN	on returning no value is provided for a target assignment
ASSIGN	the value used in target assignment
INPUT	defines an input device using a packet of the type given
COST	returns a time and power cost of executing a block
OUTPUT	defines an output device using a packet of the type given
SEND	sends the packet out over the network
ACTUATOR	defines an actuator device using given packet type
NOSPACE	used when there are no storage locations
SPACE	lists the storage locations (to help the compiler)
CLOSE	ends the module specification
NODE	provides access to the data of a node indexed by registers
PC	allows the user to modify the contents of the program counter

2.2.7 Command Parameters

size	number of parameters (triples)
typename	name of the Module being specified
basename	base class name to inherit from
packet	a packet type with which to form a base class
baseargs	base parameter list, (using any of p[1,size] in braces)
datatype	declares parameter type; const char *; float; Word and more
identifier	name of parameter that value will be constant once given
default	default setting for parameter (must match datatype's type)
op	string name of an instruction
hook	name of a block of code defining a template instruction
var_x	operand type; imm (immediate number); val (memory value); set (return value assigned to memory); nul (not used)
cost	amount of time and power taken for variable length blocks
dataname	name of a storage location
datasize	size in words of a storage location
arg_x	chosen function argument names

2.3 Connection Model

A connection model, as described in (*Connectivity Overview*, 2.1.1) is essentially responsible for determining which nodes are connected at a point in time. The primary data structure is a number of directed graphs.

It may hold environmental or system global data to make its decisions and can allow nodes to interact with their environment. A number of other functions may be instantiated such as the pre-loading of message queues and specifying how results are streamed out of an experiment.

The C++ programming interface is described in section (*Programming Interfaces*, A).

2.3.1 Note

Please note that some of the more mundane parts of this section are being rewritten. In the meantime - the examples should make the remaining omissions obvious - such as how to define a packet type.

3 Examples

3.1 Introduction

This section shows how to create experiments using BiSim by example. Basic C++ skills or general object-oriented programming knowledge is assumed. Although other targets are supported, this document refers to the Linux/bash environment and g++ 3.x.

3.2 Preparation

Firstly, we need to obtain BiSim's source code and run the provided scripts to compile the libraries for the particular target machine you want to work on. This only needs to be done once.

The stable releases for BiSim can be found in `/home/s9716341/bisim/`. Copy the latest release, e.g. `v1-07`, into your working space. The file `v1-xx/CONTENTS.TXT` describes the directory contents.

- Place the `v1-xx/scripts` directory in your path.
- Next, change to the `v1-xx/platform/run` directory.
- Run `./libraries` to compile the library files.
- Finally, type `./publish` to export the result.

The compiled binaries and required includes have now been created in the `v1-xx/pub` directory.

Before we move on to the examples, please note that the source code for these is also provided, if you would prefer not to type the code in. Sections with a border around them correspond to example source code.

3.3 Examples

3.3.1 Processor

In this example we will make an experiment to simulate a MipsR3000 processor using a supplied module. This will demonstrate some of the basics needed to get a BiSim VM experiment up and running on its own and allow us to show a few other features along the way.

Inside the `experiment` sub-directory, create a directory `processor` with the following structure:

```
experiment/          => experiment definitions
    processor/       => our current example
        bin/         => executables
        run/         => compilation scripts
        src/         => our example's source code
```

Inside the `src` directory, create a header file for our experiment, `Processor.hpp`, with the following few lines of code:

```
#include "../command/BiSim.hpp"                  // Includes the main BiSim header.

class Processor : public Engine::Connectivity    // Declare connectivity model type.
{
    public:
        Processor() { }
        void arrange(Composition &);             // Will provide network composition.
        static Processor * create(Setup);        // Will create and configure a model.
};
```

In this short demonstration, our network composition is trivial - a single processor. Also, we do not need to perform any configuration on this model. Make a file called `Processor.cpp` and type the following:

```
#include "Processor.hpp"                         // Includes the header file we just made.

void Processor::arrange(Composition &content)    // The arrangement - a node of type "pu".
{                                                //
    content[0] = Symbol("pu");                   //
}                                                //

Processor *Processor::create(Setup)              // Creates a processor model.
{                                                //
    return new Processor();                      //
}                                                //

BISIM_STARTUP(Processor)                         // Links example code to the libraries.
BISIM_EXECUTABLE                                 //
```

That's it for coding this experiment's structure. To compile it, make a file, `make_processor`, inside `run` and type:

```
g++ -o../bin/processor.exe -L../../../platform/pub/lib \
    -I../../../platform/pub/inc/base ../src/Processor.cpp -lbisim
```

This places our executable, `processor.exe`, inside our `bin` directory, tells the compiler where to find BiSim's library and header files and adds in the experiment that we just defined. On Linux/Unix you may need to give yourself permission to execute the script: `chmod u+x make_processor`. Now we can run the script by typing `./make_processor` from inside `run`. Inside our `bin` directory, we can run our executable by typing `./processor.exe`. This produces the following message:

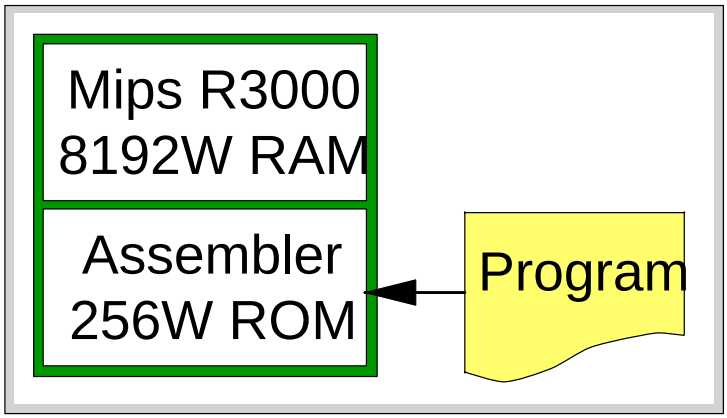
```
BiSim v1.07 Usage: ./processor.exe OPTION

  --on-line-manual         display on-line manual and user guide
  --run-experiment <file>  run experiment using script settings
  --script-default         make default script for this build
  --script-summary <file>  check and summarise script file
  --module-summary         show available modules
  --plugin-summary         show available plugins
  --machine-custom <file>  show user defined machine layouts
  --machine-intern         show internal machine layouts
  --machine-source <file>  show compiled source codes
  --machine-format <file>  show overview of machine compositions
  --suites-verbose         run all test suites in verbose mode
  --suites-summary         show summary of running all tests

The experiment "processor" is built into this program.
```

If we type: `./processor.exe --script-default > settings.txt` this places the some default settings for our experiment inside `settings.txt`. All of the settings in this file are defined by BiSim's engine, but we'll see how to add our own specialised ones in later. Open up the `settings.txt` file and, as an example, scroll down to the `Network` section:

```
[Network] ##################
                          #
graphics     0            # Natural - view - graphics stream (*)
refresh      1            # Natural - steps per connection update (*)
steps        10000        # Natural - number of steps to simulate (*)
view_time    1            # Natural - view - current time point (*)
                          #
############################
```

Modify `steps` to 200 and scroll down to the end of the file. Earlier, we stated that we wanted to create a single VM type "pu". We can define this type in this way:

```
[pu] ##########################################
                                              #
$ MipsR3000(mips,8192)                        #
$ Assemble(prog,sample.asm,256)               #
                                              #
##############################################
```



This states that we want to link the supplied `MipsR3000` and `Assemble` modules into our VM type. These modules are defined in `platform/src/module`. We call them `mips` and `prog` respectively. The `mips` definition describes a simple mips instruction set and accepts one parameter to specify its RAM size as `8192` words. The `prog` definition describes an assembly code reader that will load a file called `sample.asm` and have a limit of `256` words of memory. Please note that the parser does not expect to see spaces between arguments. Now that we have described our VM, we can specify the code that we want it to run in `sample.asm`. Again, please avoid adding spaces between instruction arguments.

```
.MODEL BISIM_VM                 ; States the target type as a BiSim VM.

.CODE                           ; Starts the program code section.

    mips-addi    step,$0,3      ; Uses mips add immediate on "step".

    loop:                       ; Defines a label.

    mips-add     $8,$8,step     ; Increments register $8 with "step".
    mips-j       loop           ; Jumps back to our label.

.DATA                           ; Starts the data section.

    step         DW 1           ; Defines the variable "step".
```

We can now run our experiment by typing: `./processor.exe --run-experiment settings.txt`. The output produced will describe a number of details about the experiment, such as the way in which the VM has been compiled and the available module types and their parameters. After this, an instruction level execution trace will be produced:

```
time: 00000001

/=============================================================================\
| node |   pc | module  com |      alpha     beta   define   |   result | batt |
|=============================================================================|
| 23b0 | 2021 |   mips addi | [00000001]  00000003  | [202d] | 00000003 | fffe |
\-----------------------------------------------------------------------------/

time: 00000015

/=============================================================================\
| node |   pc | module  com |      alpha       beta   define |   result | batt |
|=============================================================================|
| 23b0 | 2025 |   mips  add | [00000009] [0000202d] | [0009] | 00000003 | fffd |
\-----------------------------------------------------------------------------/

...
```

The first trace line shows that our node is executing the *add immediate* instruction from the *mips* module. By default, BOSim produces a data area layout for each virtual machine, the field for *pc*, the program counter refers to this address space. If any of the three instruction arguments, *alpha, beta* and *define* is wrapped in square brackets this also means that a memory index is being performed, otherwise this field will refer to immediate values. The nodes simulation internal identifier is shown as is its simulated battery level that decrements each time some power is used by a node.

We can, for example, control whether the `time` and the execution `run` is displayed by modifying the settings file entries `Network/view_time` and `Machine/view_run`. That's the end of the first example - if you would like to see how the instruction set was defined, take a look at the module declared in `platform/src/module/MipsR3000.hpp`. In the next example, we'll make our own modules.

3.3.2 Leader Election

In this more detailed experiment, we will run a simple leader election algorithm across a static network of nodes. We will implement `Modem` and `Counter` modules and a number of features will be introduced, such as data streaming, plug-ins, exceptions and settings. Inside the `experiment` sub-directory, create a directory `leader` with the following structure:

```
experiment/          => experiment definitions
    leader/          => our current example
        bin/         => executables
        run/         => compilation scripts
        src/         => our example's source code
```

Firstly, a quick overview of the algorithm used to set things in context. When a node believes that it may be a leader in its vicinity, it will periodically broadcast its own randomly chosen local identifier. If a node receives an identifier that is higher that its own, then it will consider the sender of the higher identifier its leader. If a period of time passes when no identifier is received by a node that is higher than its own, it will consider itself a leader.

Inside the `src` directory, create a file, `Counter.hpp`. We are going to declare a simple module that has only two instructions; to reset and decrement a counter. The reset function will set the counter to a random number within a specified limit. An extra register, `boss`, will be declared here that will be used to hold the local address of a leader later on.

```
MODULE(2)                                              // This module has two parameters.
    (                                                  //
        Counter,      BASE(Unconnected), (p1),         // 1. No connectivity abilities are used.
        const char *, name,              "timer"       // 2. A name-tag identifier.
    )
    INTERFACE                                          // This section defines instructions.
        //      name    fun   op0  op1  op2  time  pow  //
        //      ----    ---   ---- ---- ---- ----- ---  //
        DECLARE("init", init, val, nul, nul, 0x04, 1)  // 1. Reset using op0 contents as limit.
        DECLARE("step", step, set, nul, nul, 0x02, 1)  // 2. Decrement, set op0 if finished.
    STORAGE
        DATA(tick, 1)                                  // Data area, "tick", uses 1 word.
        DATA(boss, 1)                                  // Data area, "boss", uses 1 word.
    FUNCTIONS
        INSTRUCT(Counter, init, lim, _a, _b)           // Reset: "lim" is max counter value.
            NODE[self.tick] = Base::Random::value(lim);//    Set "tick" to random number.
            NOASSIGN                                   //    No additional assignment.
        INSTRUCT(Counter, step, _d, _a, _b)            // Decrement:
            if(NODE[self.tick] == 0) ASSIGN(1)         //    If finished, return 1.
            NODE[self.tick]--;                         //    Decrement the "tick" register.
            ASSIGN(0)                                  //    Return 0.
    SPACE tick, boss;                                  // Declares the two data areas used.
CLOSE(Counter)                                         // Ends the "Counter" module declaration.
```

That's the end of the first module declaration. The next module we are going to define is a modem so that our nodes are able to communicate. This is a more substantial component than our previous module.

We will now create a bi-directional i/o device with cyclic queues. In a file called, Modem.hpp, type:

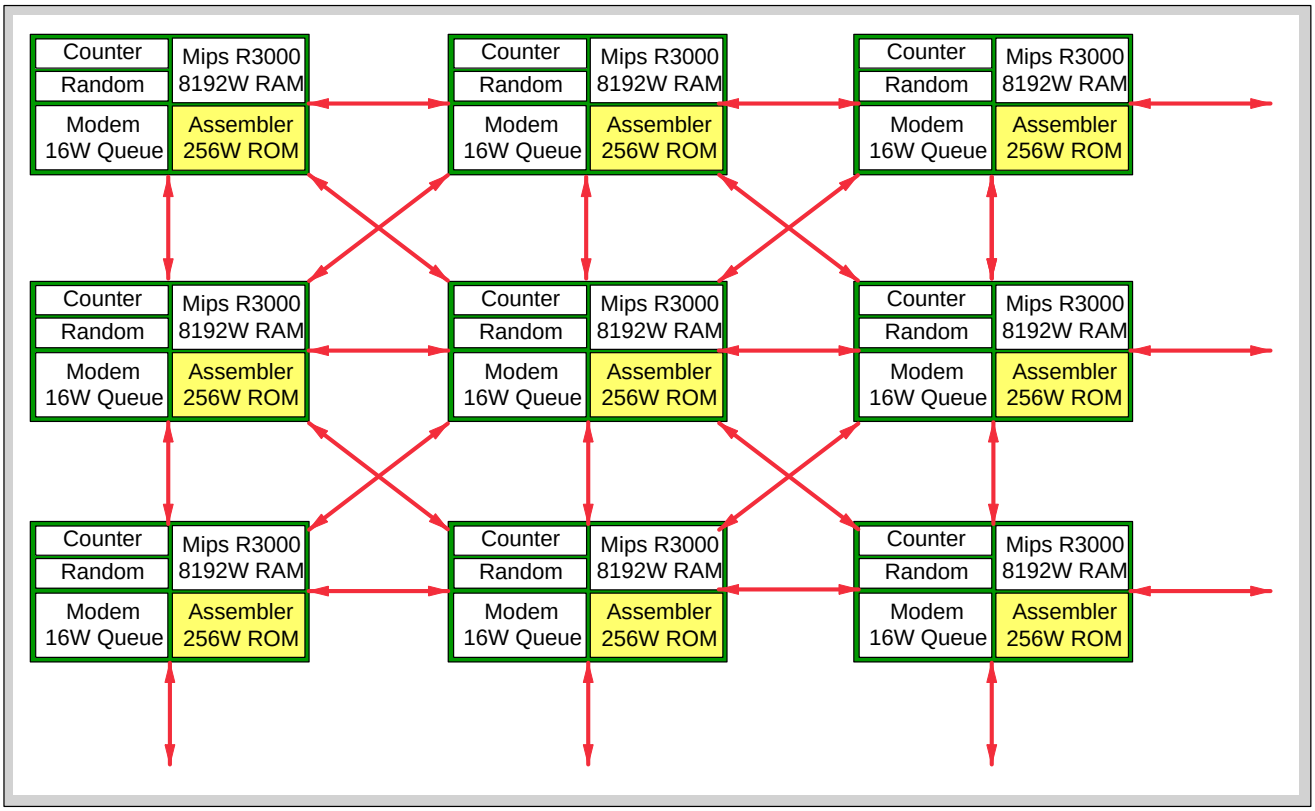
```
MODULE(3)                                          // This module has three parameters.
    (                                              //
        Modem,        BASE(Signal), (p1),          // Its packet type is "Signal".
        const char *, name,         "modem",       // A name-tag identifier.
        int,          q_size,       16             // Cyclic queue sizes in words.
    )
    INTERFACE                                      // This section defines instructions.
        DECLARE("tx",  tx,  val, nul, nul, 0x06, 1) // 1. Transmits op0 contents.
        DECLARE("txi", tx,  imm, nul, nul, 0x04, 1) // 2. Transmits op0 constant.
        DECLARE("rx",  rx,  set, nul, nul, 0x06, 1) // 3. Receives word to op0.
    STORAGE
        DATA(iHead, 1)                             // Head position for input queue.
        DATA(iTail, 1)                             // Tail position for input queue.
        DATA(iQ,    q_size)                        // Data area for cyclic input queue.
        DATA(oHead, 1)                             // Head position for output queue.
        DATA(oTail, 1)                             // Tail position for output queue.
        DATA(oQ,    q_size)                        // Data area for cyclic output queue.
    FUNCTIONS
        INSTRUCT(Modem, tx, value, _a, _b)         // Transmit:
            Word &roHead = NODE[self.oHead];       //    Take a reference to the head.
            Word next = ((roHead + 1) % self.q_size); //    Find the next head position.
            if(NODE[self.oTail] == next) ASSIGN(0) //    If no space left, return 0.
            NODE[self.oQ(roHead)] = value;         //    Place value onto queue.
            roHead = next;                         //    Move the head position forward.
            ASSIGN(1)                              //    Return 1.
        INSTRUCT(Modem, rx, _d, _a, _b)            // Receive:
            Word &riTail = NODE[self.iTail];       //    Take a reference to the tail.
            if(riTail == NODE[self.iHead]) ASSIGN(0) //    If there is no data, return 0.
            Word value = NODE[self.iQ(riTail)];    //    Retrieve the next word.
            riTail = (riTail + 1) % self.q_size;   //    Move the tail position forward.
            ASSIGN(value)                          //    Return the popped value.
        INPUT(Signal, packet)                      // Input:
            Word &riHead = NODE[iHead];            //    Take a reference to the head.
            Word next = ((riHead + 1) % q_size);   //    Find the next head position.
            if(NODE[iTail] == next) COST(2, 4)     //    If no space, deduct costs and quit.
            NODE[iQ(riHead)] = packet.message;     //    Place value onto queue.
            riHead = next;                         //    Move the head position forward.
            COST(2, 10)                            //    Deduct time and power costs.
        OUTPUT(Signal, packet)                     // Output:
            Word &roTail = NODE[oTail];            //    Take a reference to the tail.
            if(roTail == NODE[oHead]) COST(2, 4)   //    If no data, deduct costs and quit.
            packet.message = NODE[oQ(roTail)];     //    Place the next word in a packet.
            roTail = (roTail + 1) % q_size;        //    Move the tail position forward.
            SEND(Signal, packet)                   //    Broadcast the packet.
            COST(2, 10)                            //    Deduct time and power costs.
    SPACE iHead, iTail, iQ, oHead, oTail, oQ;      // Declares the data areas used.
CLOSE(Modem)                                       // Ends the "Modem" module declaration.
```

We made use of a packet type, `Singal`, inside our modem's declaration. In this simple example, nodes only send packets to each other with their identifiers inside. This identifier was called, `message`, in the earlier code. Create a file, `Signal.hpp`:

```
PACKET_BEGIN(Signal)                                    // Names the packet type, "Signal".
    PACKET_ITEM(unsigned, message)                      // A single member, "message".
PACKET_END(Signal)                                      // Completes the declaration.
```

Now we can create the main header file for our experiment, `Leader.hpp` with the following code:

```
#include "../command/BiSim.hpp"                         // Includes the main BiSim header.

class Leader : public Engine::Connectivity              // Declare connectivity model type.
{
    public:
        typedef Base::Output     Output;                // Define some namespace abbreviations.
        typedef Engine::Graph    Graph;                 //
        typedef Engine::Register Register;              //
        typedef Engine::User     User;                  //

        Leader(const Graph::PlugIn &, unsigned nodes);  // Our constructor with two parameters.
        void arrange(Composition &);                    // Will provide network composition.
        static Leader * create(Setup);                  // Will create and configure a model.

        virtual void initialise(Routes &);              // Will declare our graph configuration.
        virtual void stream(Place, unsigned) const;     // Allows us to stream results out.

    protected:
        static const Base::Exception not_square;        // Declares an exception type.
        static Base::Setup ini_count, ini_view_results; // Declares two settings types.

        Graph    m_graph;                               // Our connectivity graph.
        unsigned m_nodes;                               // The number of nodes in the network.
        unsigned m_edge;                                // The length of an edge.
};
```



In this demonstration, we are going to connect our nodes in a static grid configuration. We will manually form connections between nodes less than two grid edges away. Connectivity models are provided to help with this, but we'll do things directly this time. We will declare two settings that will appear in the experiment's script file; `count` will allow us to change the number of nodes and `view_results` will allow us to pipe out some simple results.

```
#include "Leader.hpp"
#include "Counter.hpp"
#include "Signal.hpp"
#include "Modem.hpp"
#include "math.h"

SETUP(Leader, count,        25U, "number of nodes")
SETUP(Leader, view_results,  4U, "output - results")

EXCEPTION(Leader, not_square)

Leader::Leader(const Graph::PlugIn &plug, unsigned nodes)
:
    Engine::Connectivity(),
    m_nodes(nodes),
    m_graph(plug)
{
    m_edge = (unsigned)sqrt(m_nodes);
    if(m_edge * m_edge != m_nodes) THROW(not_square)
}

Leader *Leader::create(Setup cs)
{
    Graph::PlugIn::Argument ga;
    ga.id        = Symbol("io");
    ga.overlap   = Graph::resolve_new;
    ga.overwrite = Graph::resolve_new;

    return new Leader(
        *Graph::Types::create(cs, "LossLess", ga),
        ini_count.word()
    );
}

void Leader::initialise(Routes &graphs)
{
    graphs[Symbol("io")] = &m_graph;
}
```

```
// Includes: our experiment header.
//           our counter header.
//           our signal header.
//           our modem header.
//           standard maths.

// Node count setting, default is 25.
// View results setting, 4 = to file.

// Exception if count is not square.

// Constructor:
//
//    Initialise base class.
//    Set number of nodes.
//    Set our graph's plug-in.
//
//    Find our grid's edge size.
//    If count is not square, complain.


// Creating our "leader" model:
//
//    Instantiate a plug-in:
//       Call it "io".
//       Overlapping Txs interfere.
//       Multiple inputs interfere.
//
//    Creating the "leader" object:
//       Use "LossLess" graph plug-in.
//       Use the settings node count.



// Initialise Routes:
//
//    We provide our graph, "io".
```

```
void Leader::arrange(Composition &content)
{
    int i;
    Graph::Info side(10, 1);
    Graph::Info diag(14, 1);

    for(unsigned r=0; r<m_nodes; r++) content[r] = Symbol("lead");

    for(unsigned x=0; x<m_edge; x++) {
        for(unsigned y=0; y<m_edge; y++) {
            bool l = (x > 0);
            bool r = (x + 2 < m_edge);
            bool t = (y > 0);
            bool b = (y + 2 < m_edge);
            i = x+y*m_edge;
            if(l)   m_graph.connect(i, (x-1) + (y  ) * m_edge, side);
            if(r)   m_graph.connect(i, (x+1) + (y  ) * m_edge, side);
            if(t)   m_graph.connect(i, (x  ) + (y-1) * m_edge, side);
            if(b)   m_graph.connect(i, (x  ) + (y+1) * m_edge, side);
            if(l&t) m_graph.connect(i, (x-1) + (y-1) * m_edge, diag);
            if(l&b) m_graph.connect(i, (x-1) + (y+1) * m_edge, diag);
            if(r&t) m_graph.connect(i, (x+1) + (y-1) * m_edge, diag);
            if(r&b) m_graph.connect(i, (x+1) + (y+1) * m_edge, diag);
        }
    }
}

void Leader::stream(Place p, unsigned time) const
{
    if(p != place_end) return;

    static Register s_boss = User::open().spy("lead", "tick", "boss");
    static const Engine::Network &s_network = Engine::Network::open();

    Output::Channel ch = Output::open(ini_view_results);

    for(int i=0; i<m_nodes; i++) {
        ch("[%8.8x]", s_network.node(i)[s_boss]);
        if(0 == ((i+1) % m_edge)) ch("\n");
    }
    Output::flush(ini_view_results);
}

BISIM_STARTUP(Leader)
BISIM_EXECUTABLE
```

```
// Declaring our arrangement:
//
//    Temporary node index.
//    Edge delay 10, b/w 1.
//    Edge delay 14, b/w 1.
//
//    Make "m_nodes" nodes.
//
//    For each node (x,y):
//
//       can go left?
//       can go right?
//       can go up?
//       can go down?
//       calculate index.
//       join left.
//       join right.
//       join up.
//       join down.
//       join left-top.
//       join left-down.
//       join right-top.
//       join right-down.


// Streaming out data:
//
//    Wait to the end.
//
//    Spy on register "boss".
//    Take reference.
//
//    Open output channel.
//
//    For each node:
//       Write out "boss".
//       Add formatting.
//
//    Flush the data stream.


// Links our example code.
//
```

And that's the end of our coding for this experiment's structure. We can compile it, like before, using a script file, `make_leader`, inside the `run` directory:

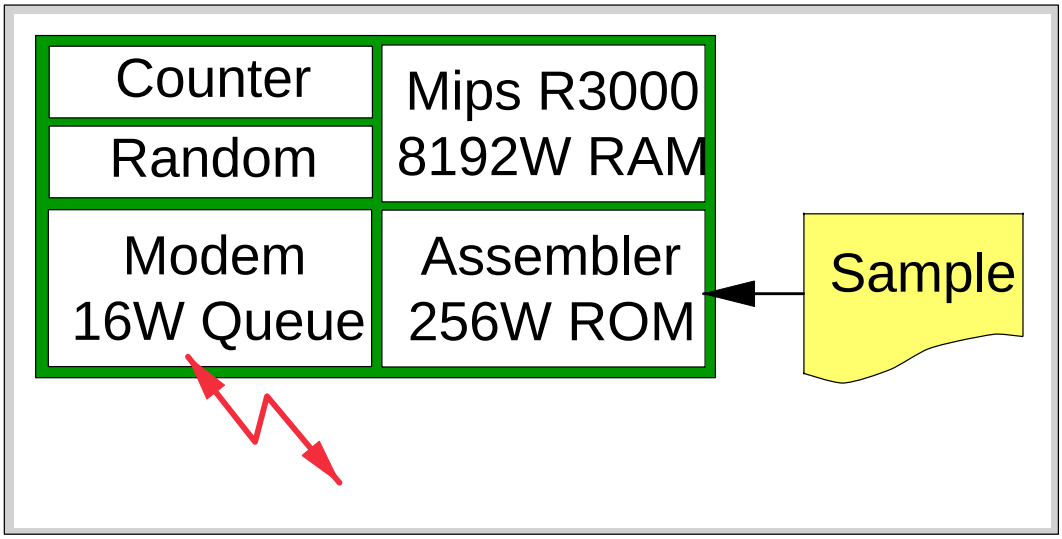
```
g++ -o../bin/leader.exe -L../../../platform/pub/lib \
    -I../../../platform/pub/inc/base ../src/Leader.cpp -lbisim
```

And again, if we change into the `bin` directory and type: `./leader.exe --script-default > settings.txt` this produces our default script file. This file will contain the two settings that we defined earlier, in addition to BOSim's usual settings:

```
[Leader] ##################
                          #
count        25           # Natural - number of nodes (*)
view_results 4            # Natural - output - view results (*)
                          #
############################
```

We can open up this file and add our VM composition, called `lead`:

```
[lead] ##########################
                                 #
$ Modem(io,16)                   #
$ Counter(tick)                  #
$ MipsR3000(mips,1)              #
$ Random(rand)                   #
$ Assemble(prog,sample.asm,128)  #
                                 #
###################################
```



This creates a VM having; a `Modem` with cyclic queue sizes of `16` words, a `Counter`, a `MipsR3000` instruction set with `1` word of RAM, a `Random` number generator and `Assemble` providing a means to read assembly code from `sample.asm` into a limit of `128` bytes of memory. Next, we will program our new VM in the file `sample.asm`.

```
.MODEL BISIM_VM                                 ; States the target type as a BiSim VM.

.CODE                                           ; Starts the program code section.

initialisation:                                 ; Initialisation section:

    mips-addi   long_wait,$0,20                 ; Assign 20 to our "long_wait" count.
    mips-addi   short_wait,$0,3                 ; Assign 3 to our "short_wait" count.
    rand-geni   my_id,1000000                   ; Generate a random id for myself.
    tick-init   long_wait                       ; Initialise counter with "long_wait".

start_loop:                                     ; Start Loop section:

    io-rx       message                         ; Receive a message.
    mips-slt    continue,my_id,message          ; Is its value higher than my id?
    mips-beq    continue,$0,check_timeout       ; If not, skip to "check timeout".

    mips-add    boss,$0,message                 ; Set my boss to be the message value.
    tick-init   long_wait                       ; Initialise counter with "long_wait".

check_timeout:                                  ; Check Timeout section:

    tick-step   continue                        ; Count down. Has the counter run out?
    mips-beq    continue,$0,start_loop          ; If not, skip to "start loop".

    mips-add    boss,$0,my_id                   ; Set my boss to be myself.
    io-tx       my_id                           ; Transmit my id.
    tick-init   short_wait                      ; Initialise counter with "short_wait".
    mips-j      start_loop                      ; Go back to "start loop".

.DATA                                           ; Starts the data section.

    my_id       DW 1                            ; My random identifier.
    message     DW 1                            ; Place to store a message.
    skip        DW 1                            ; A flag.
    continue    DW 1                            ; A flag.
    long_wait   DW 1                            ; Long wait duration.
    short_wait  DW 1                            ; Short wait duration.
```

We can now run our experiment by typing `./leader.exe --run-experiment settings.txt`. A simple table of results is produced in `results.txt` - have a quick look inside this file. Results are categorised by the class that produced them. We can test our exception out too by making the setting, `Leader/count`, a non-square number such as `24`. Doing so will produce the output:

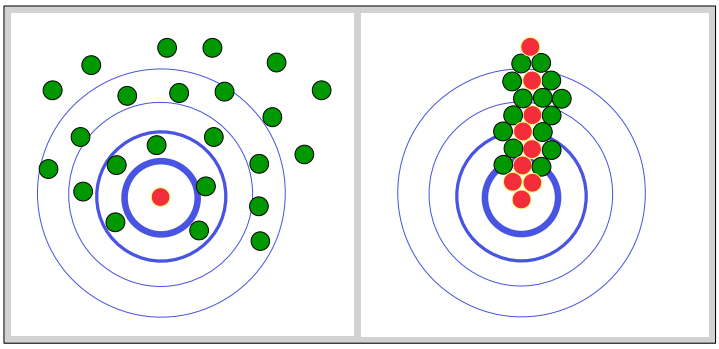
```
BiSim v1.07 Fatal Exception Handler:

    --> Exception:   Leader::not_square
    --> Source Code: ../src/Leader.cpp (line 19)
    --> Function:    Leader
```

Well, nice as plain text results are, it may be helpful to have an idea of what's going on while an experiment is running. In our next example we will look at using the simple visualisation tool and using the supplied `Grid2D` connectivity model to simulate a fluid network.

3.3.3 Aggregating Amoebae

This is a more substantial example than the previous two and will illustrate some more advanced features of BȦSim.

But first, lets set the scene. Deep below the surface of the soil wander dictiostylium amoebae eating nutrients to stay alive. If some of these amoebae start to run out of food, they will broadcast a diffuse signal to indicate so. Other amoebae in the vicinity detect this signal and move towards its source by detecting a chemical concentration difference across their cell membrane, which indicates the direction of the source. This deceptively naive mechanism not only causes these cells to clump together into smaller, then larger groups, but also causes clumped cells, which join in the broadcasting, to synchronise is such a manner that they wriggle together as a slug made up of amoebae in search of a more nutrient-plentiful area. This example will cover aspects of the clumping behaviour.



We will implement two modules to simulate an amoeba's message signaling `IO` mechanism and its behavioural `Control`. Two examples will be given to demonstrate user-defined plugins that can specify the way in which messages are transmitted by BȦSim's engine. A fluidly moving network of amoebae will be simulated using the `Grid2D` supplied module. Finally, we will hook our example up to a simple visualisation tool and watch our amoebae do their stuff.

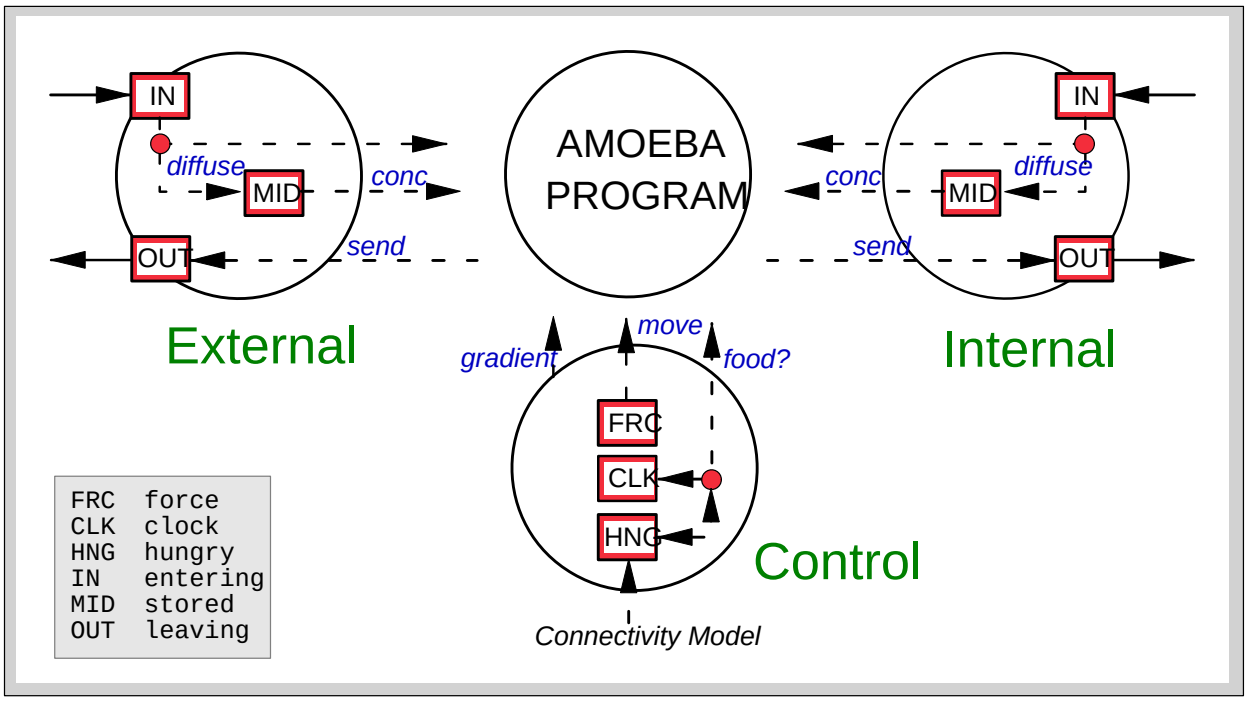
As usual, inside the `experiment` sub-directory, create a directory, `amoeba` with the following structure:

```
experiment/          => experiment definitions
    amoeba/          => our current example
        bin/         => executables
        run/         => compilation scripts
        src/         => our example's source code
```

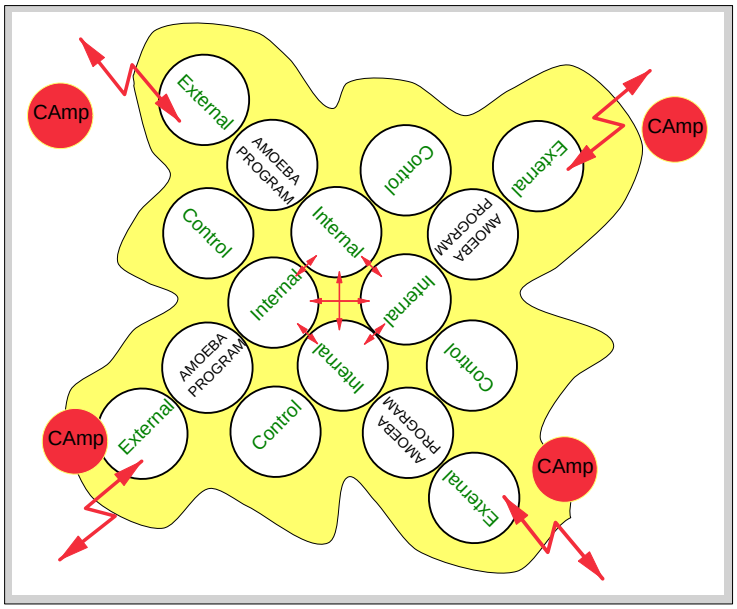
The name of the chemical message used for these amoebae's signaling is called CAmp. Create a file, `CAmp.hpp`, to describe this type of packet:

```
PACKET_BEGIN(CAmp)                                  // Names the packet type, "CAmp".
    PACKET_ITEM(unsigned, concentration)            // A single member, "concentration".
PACKET_END(CAmp)                                    // Completes the declaration.
```

We will split an amoeba up into four compartments so that we are able to detect a message concentration gradient across it. The picture below shows an amoeba split into four components, each with an external and internal `IO` module that simulate intra-cell (cell boundary) and inter-cell (inside a cell) communication respectively. The `Control` module will provide a means to coordinate an amoeba compartment's actions and the program module will allow us to try out different possible behaviours.



The following picture can be referred to while the module definitions are being presented next. It shows how the modules in each compartment interface together. All communication signals are in the form of `CAmp` molecules and the internal and external components are both types of `IO` module.



The diffusion coefficient used in the IO module definition below determines a molecule's tendency to stay in one compartment rather than to diffuse out of it. We can create an amoeba compartment's IO module in a file called IO.hpp that will diffuse the CAmp message:

```
MODULE(3)                                                   // This module has three parameters.
    (                                                       //
        IO,           BASE(CAmp), (p1),                     // Its packet type is "CAmp".
        const char *, name,       "io",                     // A name-tag identifier.
        float,        retain,     0.3F                      // Retention coefficient.
    )
    INTERFACE                                               // This section defines instructions.
        DECLARE("send", send, imm, nul, nul, 0x04, 1)       // 1. Transmit a fixed molecule no..
        DECLARE("pass", send, val, nul, nul, 0x04, 1)       // 2. Transmit variable molecule no..
        DECLARE("fuse", fuse, set, nul, nul, 0x04, 1)       // 3. Molecule diffusion into part.
        DECLARE("conc", conc, set, nul, nul, 0x04, 1)       // 4. Determines CAMP concentration.
    STORAGE
        DATA(leaving, 1)                                    // Molecules diffusing outwards.
        DATA(entering, 1)                                   // Molecules diffusing inwards.
        DATA(stored, 1)                                     // Molecules inside compartment.
    FUNCTIONS
        INSTRUCT(IO, send, inward, _a, _b)                  // Diffuse outwards:
            NODE[self.leaving] += inward;                   //    Increment molecules leaving.
            NOASSIGN                                        //    No additional assignment.
        INSTRUCT(IO, fuse, _d, _a, _b)                      // Diffuse inwards:
            float value = NODE[self.entering] * self.retain;//    How many molecules leave?
            Word out = (Word)(value);                       //    Round this to a whole number.
            NODE[self.stored]  += (NODE[self.entering] - out);//  Add the remainder to out store.
            NODE[self.entering] = 0;                        //    Reset the inwards count.
            ASSIGN(out)                                     //    Return how many left.
        INSTRUCT(IO, conc, _d, _a, _b)                      // Concentration Query:
            Word value = NODE[self.stored];                 //    Retrieve the CAMP conc..
            NODE[self.stored] = 0;                          //    Reset the CAMP conc..
            ASSIGN(value)                                   //    Return the concentration.
        INPUT(CAmp, molecule)                               // Input:
            NODE[entering] += molecule.concentration;       //    Add molecules to incoming.
            COST(1, 1)                                      //    Arbitrary power and time cost.
        OUTPUT(CAmp, molecule)                              // Output:
            if(NODE[leaving] > 0) {                         //    If some molecules are leaving.
                molecule.concentration = NODE[leaving];     //    Place amount in packet.
                NODE[leaving] = 0;                          //    Reset the number leaving.
                SEND(CAmp, molecule)                        //    Broadcast the packet.
            }                                               //
            COST(1, 1)                                      //    Arbitrary power and time cost.
    SPACE leaving, entering, stored;                        // Declare the data areas used.
CLOSE(IO)                                                   // Ends the "IO" module declaration.
```

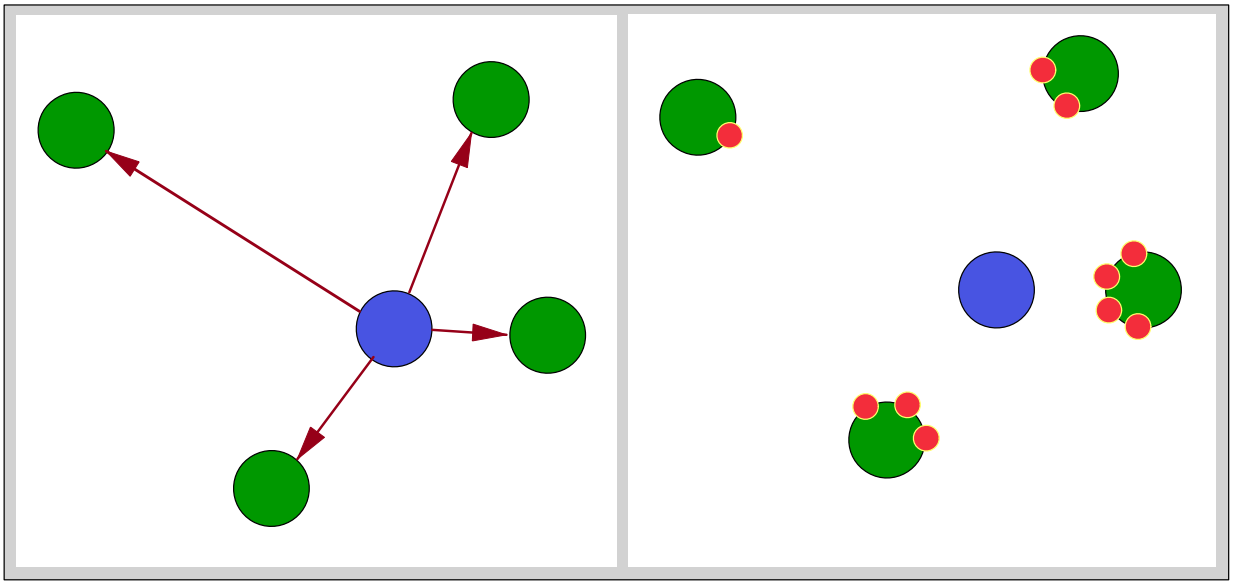
We can now define a second type of message that will interface with the connectivity model code we will be coming to later. This message will provide information on whether an amoeba is lacking nutrients, and will also state whether a force is being exerted on an amoeba compartment's cell boundary that may result in subsequent movement.

```
PACKET_BEGIN(CellState)                    // Names the packet type, "CellState".
    PACKET_ITEM(unsigned, hungry)          // Specifies whether cell is hungry.
    PACKET_ITEM(unsigned, force)           // Is there a cell boundary force?
PACKET_END(CellState)                      // Completes the declaration.
```

This takes us on to our second module. Make a file called `Control.hpp`. This module will simulate the control of a compartment of an amoeba. The threshold described below refers to the required gradient difference between internal and external signal concentrations to cause a force to be exerted on a cell boundary.

```
MODULE(4)                                                   // This module has four parameters.
    (                                                       //
        Control,      BASE(CellState), (p1),                // 1. It communicates using "CellState".
        const char *, name,            "io",                // 2. A name-tag identifier.
        int,          cycle,           1,                   // 3. "I'm hungry" broadcast period.
        float,        threshold,       0.8F                 // 4. Its gradient threshold.
    )
    INTERFACE                                               // This section defines instructions:
        DECLARE("food", food, set, nul, nul, 0x04, 1)       // 1. A hungry cell shouts.
        DECLARE("move", move, nul, nul, nul, 0x04, 1)       // 2. A compartment exerts a force.
        DECLARE("grad", grad, set, val, val, 0x04, 1)       // 3. Internal/external CAmp gradient.
    STORAGE
        DATA(hungry, 1)                                     // Whether this cell is hungry.
        DATA(clock, 1)                                      // A crude local timer.
        DATA(force, 1)                                      // Whether a force is being exerted.
    FUNCTIONS
        INSTRUCT(Control, food, _d, _a, _b)                 // Hunger:
            if(++NODE[self.clock] > self.cycle) {           //    If its time to check our hunger.
                NODE[self.clock] = 0;                       //    Then reset our counter.
                if(NODE[self.hungry]) ASSIGN(1)             //    If we are hungry, return 1.
            }                                               //
            ASSIGN(0)                                       //    Otherwise return 0.
        INSTRUCT(Control, grad, _d, external, internal)     // Gradient:
            Word offset = (Word)(external * self.threshold);//    Offset ext. by threshold mult..
            ASSIGN(internal > external)                     //    Does int. exceed offset ext.?
        INSTRUCT(Control, move, _d, _a, _b)                 // Move:
            NODE[self.force] = 1;                           //    Exert a force on our wall.
            ASSIGN(0)                                       //    No additional assignment.
        INPUT(CellState, state)                             // Input:
            NODE[hungry] = state.hungry;                    //    Update our hunger status.
            COST(0x4, 1)                                    //    Arbitrary costs.
        ACTION(CellState, state)                            // Actuator:
            if(NODE[force]) {                               //    If there is a force.
                if(NODE[hungry] == 0) {                     //    And if we are not hungry.
                    state.force = 1;                        //    State that there is a force.
                    SEND(CellState, state)                  //    And tell the connectivity model.
                    NODE[force] = 0;                        //    Reset our force flag.
                }                                           //
            }                                               //
    SPACE hungry, clock, force;                             // Declares the data areas used.
CLOSE(Control)                                              // Ends the "Control" module.
```

It is possible to define a set of methods to specify message delivery delays and channel bandwidths for a connectivity model and tell the BŝSim engine to use one or a number of them from its interpreted script file. This kind of specialisation is called a plug-in here. As well as being able to define a plug-in for a built-in BŝSim feature, we'll also show later on in this example how we can create our own types of plug-ins.



We would like to be able to adjust the model of message passing so that the time it takes a molecule to travel between two cells, or compartments, is proportional to the distance. Also, in this instance, we need to split up the number of molecules received by each neighbour : We specify that the closer to a message source a cell is, the higher share of a particular molecule broadcast it will receive. The supplied `Grid2D` connectivity model provides the slot into which we can add our plug-in definition. We can define a header file, `Proportional.hpp`:

```
#include "../model-2D/Grid2D.hpp"                                  // Include the Grid2D model.

struct Proportional : public Model::Grid2D::PlugIn                 // We specialise Grid2D's plug-in.
{                                                                  //
    Proportional(const Argument &);                                // Configured with an argument.
    unsigned delay(const Distances &, const Distance &) const;     // Declare delay method.
    unsigned bandwidth(const Distances &, const Distance &) const; // Declare bandwidth method.
};

REGISTER_PLUGIN(Proportional, Model, Grid2D)                       // Link the plug-in to BiSim.
```

And the corresponding source code, in `Proportional.cpp`:

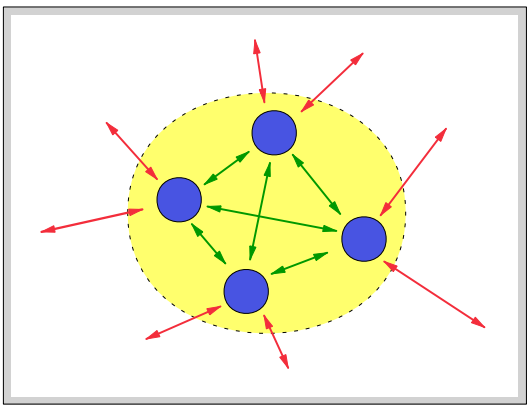
```
#include "Proportional.hpp"

Proportional::Proportional(const Argument &arg)
:
    PlugIn(arg)
{ }

unsigned Proportional::delay(
    const Distances &s, const Distance &d) const
{
    return (unsigned)(d);
}

unsigned Proportional::bandwidth(
    const Distances &s, const Distance &d) const
{
    float count = 0;
    unsigned size = s.size();
    if(size == 1) return 1000;
    for(unsigned r=0; r<s.size(); r++) count += s[r];
    if(count == 0) return 0;
    float result = (((2.0 / size) - (d / count)) * 1000);
    return (result > 0) ? (unsigned) result : 0;
}
```

```
// Include the header we just made.

// Our class constructor:
//    The argument is inherited from
//    the derived plug-in class and so
//    must be passed on to it.

// Determining delay:
//    We only need to know the distance
//    in question, d, not the set of
//    all connected cell distances.


// Determining bandwidth:
//
//
//
//    If there is only one connection,
//    return our maximum limit, 1000.
//    Calculate the distances sum.
//    If empty, return 0.
//    Proportion bandwidth by proximity.
//    Assign positive bandwidth values.
```

It is also possible to override the way that the BiSim engine broadcasts messages, as is demonstrated in the example files `LossyCAmp.hpp` and `LossyCAmp.cpp`. In this instance, given an engine-supplied node connectivity graph, this method splits a message up into parts, that is, proportions its molecules, depending on the bandwidth between nodes.

Next we will define our connectivity model for this experiment. Our model will be derived from the `Grid2D` supplied class which provides some useful methods for manipulating flat grid based networks and is also able to integrate with the external graphics viewer. We will demonstrate how to create our own type of plug-in too.

Amoebae will be split up into four compartments, each represented as a node. The compartments will be arranged in a diamond shape and labeled as the main four points on a compass. This will allow us to detect chemical gradients. Within an amoeba *quartet*, communication will have different characteristics than it will between amoebae through the soil. For this reason, two different connectivity graphs are used, one for internal and external communication.



The map data structure used below is taken from the C++ Standard Template Library for convenience. Make a file, Amoeba.hpp:

```
#include "../command/BiSim.hpp"                                 // Includes the main BiSim header.
#include "../model-2D/Grid2D.hpp"                               // Includes the Grid2D header.

class Amoeba : public Model::Grid2D {                           // Derives our model from Grid2D.
    public:
        typedef Base::Symbol              Symbol;               // Define some namespace abbreviations.
        typedef Base::Setup               Setup;                //
        typedef Engine::Graph             Graph;                //
        typedef Engine::Packet            Packet;               //
        typedef Engine::User::Composition Composition;          //

        struct Quartet { Index north, south, east, west; };     // Four compartments.
        struct PlugIn {                                         // Our own type of plug-in.
            struct Argument { unsigned delay, bandwidth; };     // Define two parameters.
            PlugIn(const Argument &);                           // Constructor accepts parameters.
            const unsigned delay, bandwidth;                    // A place to store the data.
        };

        Amoeba(const PlugIn &, const Model::Grid2D::PlugIn &,   // Main constructor takes four plug-ins.
            const Graph::PlugIn &in, const Graph::PlugIn &ex    // One for Grid2D, one for each Graph,
        );                                                      // and the one we just defined above.
        virtual ~Amoeba();

                                                                // Virtual destructor.
        static PlugIn        * load_connectivity(Setup);        // Loads our plug-in from settings file.
        static Graph::PlugIn * load_graph(Setup);               // Loads a Graph plug-in.
        static Amoeba        * create(Setup);                   // Will create and configure a model.
        static Symbol          machine_type();                  // Specifies which VM to use.

        void add(const Quartet &);                              // Adds a four compartment amoeba in.

        virtual void initialise(Queues &);                      // Pre-loads queues before experiment.
        virtual void initialise(Routes &);                      // Declares our two graph configuration.
        virtual void update(Index, const Packets &);            // Accepts cell-wall force indicators.
        virtual void neighbours(Index, Indeces &) const;        // Calculates a cell's neighbours.
        void         arrange(Composition &);                    // Provides our network composition.
        void         forces(bool);                              // Toggles whether we use sticky forces.

        const Graph &internal() const;                          // Return our graph for inside messages.
        const Graph &external() const;                          // Return our cell-cell graph.

        static const Base::Exception invalid_group,             // Some unrecoverable error conditions.
            invalid_quartet, duplicate_index;                   //
```

```
    protected:
        typedef unsigned QId;

        typedef std::map<QId, Quartet> Quartets;
        typedef std::map<Index, QId>   Groups;

    private:
        void shuffle(Index, const Packet &);
        void adjust(const Quartet &, const Point &);
        void hook(const Quartet &);

        static Base::Setup ini_delay, ini_bandwidth,
            ini_machine, ini_graph_type, ini_graph_id,
            ini_graph_olap, ini_graph_owrite, ini_count;

        Quartets       m_quartets;
        Groups         m_groups;
        Graph          m_diamonds;
        const PlugIn & m_rcOptions;
        bool           m_forces;
};
```

```
// Define an index type for amoebae.
//
// A map of index to amoeba.
// Compartment to index map.


// Causes force propagation.
// Moves an amoeba's position.
// Add amoeba's interconnections.

// A collection of script settings.
//
//

// A collection of amoebae.
// Maps compartments to amoebae.
// Our cell-internal graph.
// Stores our plug-in data.
// Whether we use sticky forces.
```

And now on to its implementation, make a file, Amoeba.cpp:

```
#include "Amoeba.hpp"
#include "CAmp.hpp"
#include "IO.hpp"
#include "CellState.hpp"
#include "Control.hpp"

EXCEPTION(Amoeba, duplicate_index)
EXCEPTION(Amoeba, invalid_group)
EXCEPTION(Amoeba, invalid_quartet)

SETUP(Amoeba, count,        10U,        "number of amoebae")
SETUP(Amoeba, delay,        1U,         "internal amoeba transfer delay")
SETUP(Amoeba, bandwidth,    1U,         "internal amoeba bandwidth")
SETUP(Amoeba, machine,      "amoeba",   "node machine type")
SETUP(Amoeba, graph_type,   "LossLess", "Effect of signal strength on transfer")
SETUP(Amoeba, graph_id,     "graph",    "Unique graph identifier")
SETUP(Amoeba, graph_olap,   3U,         "Overlap collision")
SETUP(Amoeba, graph_owrite, 3U,         "Overwrite collision")
```

Above, we have included the relevant headers, and specified our exceptions and default settings values. The two last settings are set to option three by default, which allows messages to be broadcast and received simultaneously. That is, if two molecules diffuse within the same time slice into a compartment, we consider that they both are delivered successfully.

```
Amoeba::PlugIn::PlugIn(const Argument &arg)                  // Here we just assign two parameters
:                                                            //    to our member variables when
    delay(arg.delay), bandwidth(arg.bandwidth)               //    our plug-in is initialised.
{ }                                                          //

void Amoeba::arrange(Engine::User::Composition &content)     // Describe our network composition:
{                                                            //
    for(unsigned r=0; r<ini_count.word(); r++) {             // For each node (from settings) -
        Quartet q;                                           //    Make a quartet.
        q.north = r * 4 + 1;                                 //    Fill in identifiers.
        q.south = r * 4 + 2;                                 //
        q.east  = r * 4 + 3;                                 //
        q.west  = r * 4 + 4;                                 //
        add(q);                                              //    Add quartet to our model.
        content[q.north] = machine_type();                   //    Assign VM type to each.
        content[q.south] = machine_type();                   //
        content[q.east]  = machine_type();                   //
        content[q.west]  = machine_type();                   //
    }                                                        //
}                                                            //

void Amoeba::initialise(Queues &queues)                      // Pre-load the queues before run:
{                                                            //
    CellState state(Symbol("state"));                        // Make a cell-state message.
    state.access().hungry = true;                            // Set its "hungry" value true.
    Symbol symbol("control");                                // Label to "Control" module.
                                                             //
    queues[1][symbol].push_back(state);                      // Make the first quartet hungry.
    queues[2][symbol].push_back(state);                      //
    queues[3][symbol].push_back(state);                      //
    queues[4][symbol].push_back(state);                      //
}                                                            //

void Amoeba::initialise(Routes &graphs)                      // State that we have two graphs:
{                                                            //
    graphs[internal().id()] = &(internal());                 // Add our internal graph.
    graphs[external().id()] = &(external());                 // Add out external graph.
}                                                            //

Base::Symbol Amoeba::machine_type()                          // State our VM type:
{                                                            //
    const char *mType = ini_machine.tag().c_str();           // Get tag from settings file.
    return Symbol(mType);                                    // Return that tag as a symbol.
}                                                            //
```

Next, we will define the connectivity model's constructor, destructor and a method to sprinkle amoebae into the initial grid positions:

```
Amoeba::Amoeba(const PlugIn &options,                          // Amoeba constructor:
    const Grid2D::PlugIn &physical, const Graph::PlugIn &in,   //    Pass our parameters to
    const Graph::PlugIn &ex)                                   //    base class and member
:                                                              //    variables as usual.
    Grid2D(physical, ex), m_diamonds(in),                      //
    m_rcOptions(options), m_forces(false)                      //
{ }                                                            //

Amoeba::~Amoeba() { }                                          // Empty Destructor.

void Amoeba::add(const Quartet &q)                             // Add a quartet:
{                                                              //
    bool b0 = (m_groups.find(q.north) != m_groups.end());      // Check for duplicate ids:
    bool b1 = (m_groups.find(q.south) != m_groups.end());      //
    bool b2 = (m_groups.find(q.east)  != m_groups.end());      //
    bool b3 = (m_groups.find(q.west)  != m_groups.end());      //
                                                               //
    if(b0 || b1 || b2 || b3) THROW(duplicate_index);           // If found, throw exception.
                                                               //
    Point p, loc;                                              // Location (x,y) variables.
    bool done = false;                                         // Iteration flag.
                                                               //
    unsigned x = Grid2D::options().size.x;                     // Get grid's width.
    unsigned y = Grid2D::options().size.y;                     // Get grid's height.
                                                               //
    while(!done) {                                             // Until we are finished:
        loc = Point(                                           //    Pick a random point.
            Base::Random::value(x/3) * 3,                      //
            Base::Random::value(y/3) * 3                       //
        );                                                     //
        if(indeces(loc + Point(1, 2)).size() == 0) done = true; //    If it is not already
    }                                                          //    occupied we are done.
                                                               //
    unsigned id = m_quartets.size();                           // Obtain a quartet id.
                                                               //
    m_quartets[id] = q;                                        // Map this id to quartet.
    m_groups[q.north] = id; m_groups[q.south] = id;            // Map compartments to
    m_groups[q.east]  = id; m_groups[q.west]  = id;            //    their quartets.
                                                               //
    Grid2D::add(q.north, loc + Point(1, 2));                   // Add each compartment of
    Grid2D::add(q.south, loc + Point(1, 0));                   //    this quartet into
    Grid2D::add(q.east,  loc + Point(2, 1));                   //    the underlying grid
    Grid2D::add(q.west,  loc + Point(0, 1));                   //    connection model.
                                                               //
    hook(q);                                                   // Interconnect quartet.
}                                                              //
```

The functions below have been included for cosmetic reasons. The first two wrap up the two graphs, one held by the Grid2D sub-class and the other defined by this Amoeba class. The last function provides a means to toggle the behaviour that causes forces to propagate through a row of amoebae.

```
const Engine::Graph &Amoeba::internal() const
{
    return m_diamonds;
}

const Engine::Graph &Amoeba::external() const
{
    return Grid2D::graph();
}

void Amoeba::forces(bool value)
{
    m_forces = value;
}
```

```
// The internal graph:
//    return our sets of
//    four interconnected
//    compartments.

// The external graph:
//    return the global
//    amoeba connection
//    graph from Grid2D.

// Toggle forces
//
//    set true or false.
//
```

The methods below are concerned with amoebae compartments exerting forces on their side of the cell membrane possibly causing the amoeba to move in that direction.

```
void Amoeba::update(Index index, const Packets &packets)
{
    Packets::const_iterator ip = packets.begin();
    while(ip != packets.end()) {
        shuffle(index, (*ip));
        ip++;
    }
}

void Amoeba::shuffle(Index i, const Packet &p)
{
    Point current, offset;
    sCAmp state = CAmp::convert(p);

    Groups::iterator ig = m_groups.find(i);
    if(ig == m_groups.end()) THROW(invalid_group)
    Quartets::iterator iq = m_quartets.find((*ig).second);
    if(iq == m_quartets.end()) THROW(invalid_quartet)
    Quartet &rQ = (*iq).second;

    current = position(i);

    if(i == rQ.north) offset.y += 3;
    if(i == rQ.south) offset.y -= 3;
    if(i == rQ.east)  offset.x += 3;
    if(i == rQ.west)  offset.x -= 3;

    Point destination = current + offset;
    if(!in_range(destination)) return;

    const Indeces &rI = indeces(destination);
    if(rI.size() == 0) { adjust(rQ, offset); }
    else { if(m_forces) shuffle((*rI.begin()), p); }
}

void Amoeba::adjust(const Quartet &q, const Point &offset)
{
    move(q.north, position(q.north) + offset);
    move(q.south, position(q.south) + offset);
    move(q.east,  position(q.east)  + offset);
    move(q.west,  position(q.west)  + offset);
}
```

```
// Update notification:
//
// For each update:
//    Shuffle the sender
//
//
//
//

// Shuffling amoebae:
//
// Location (x,y) variables.
// Access the message.
//
// Locate the sender.
// Consistency check.
// Check sender's group.
// Consistency check.
// Access the group.

// Obtain groups place.

// Calculate move offset.
//
//
//

// Calculate destination.
// Check its within grid.

// Is destination free?
// If so: adjust position.
// If not: propagate force.
//

// Adjust a group:
//
// Move each compartment.
// Using Grid2D's "move".
//
//
//
```

The following two functions form the interconnections inside an amoeba between its compartments and calculate the neighbours of a cell.

```
void Amoeba::hook(const Quartet &q)
{
    Graph::Info info(m_rcOptions.delay, m_rcOptions.bandwidth);

    m_diamonds.connect(q.north, q.south, info);
    m_diamonds.connect(q.north, q.east,  info);
    m_diamonds.connect(q.north, q.west,  info);

    m_diamonds.connect(q.south, q.north, info);
    m_diamonds.connect(q.south, q.east,  info);
    m_diamonds.connect(q.south, q.west,  info);

    m_diamonds.connect(q.east,  q.north, info);
    m_diamonds.connect(q.east,  q.south, info);
    m_diamonds.connect(q.east,  q.west,  info);

    m_diamonds.connect(q.west,  q.north, info);
    m_diamonds.connect(q.west,  q.south, info);
    m_diamonds.connect(q.west,  q.east,  info);
}

void Amoeba::neighbours(Index index, Indeces &i) const
{
    Grid2D::neighbours(index, i);

    Groups::const_iterator ig = m_groups.find(index);
    if(ig == m_groups.end()) THROW(invalid_group)
    Quartets::const_iterator iq = m_quartets.find((*ig).second);
    if(iq == m_quartets.end()) THROW(invalid_quartet)
    const Quartet &rQ = (*iq).second;

    if(i.end() != i.find(rQ.north)) i.erase(rQ.north);
    if(i.end() != i.find(rQ.south)) i.erase(rQ.south);
    if(i.end() != i.find(rQ.east))  i.erase(rQ.east);
    if(i.end() != i.find(rQ.west))  i.erase(rQ.west);
}
```

```
// Form interconnections:
//
// Get graph settings.
//
// North nodes connections.
//
//
//
// South nodes connections.
//
//
//
// East nodes connections.
//
//
//
// West nodes connections.
//
//
//

// Find external neighbours:
//
// Algorithm description:
//
//    We use the Grid2D
//    method to work out a
//    node's neighbours,
//    except we say that
//    local interconnections
//    are not counted in
//    this external graph.
```

Finally, we will add the code that deals with loading our settings and plug-ins that we are using.

```
Engine::Graph::PlugIn *Amoeba::load_graph(Setup error)
{
    const char *gType = ini_graph_type.tag().c_str();
    const char *gId   = ini_graph_id.tag().c_str();

    Graph::PlugIn::Argument ga;
    ga.id = Base::Symbol(gId);
    ga.overlap = (Graph::Resolve)(ini_graph_olap.word());
    ga.overwrite = (Graph::Resolve)(ini_graph_owrite.word());

    return Graph::Types::create(error, gType, ga);
}

Amoeba::PlugIn *Amoeba::load_connectivity(Setup)
{
    Amoeba::PlugIn::Argument arg;
    arg.delay = ini_delay.word();
    arg.bandwidth = ini_bandwidth.word();
    return new PlugIn(arg);
}

Amoeba *Amoeba::create(Setup cs)
{
    Grid2D::PlugIn *pP   = Grid2D::load_connectivity(cs);
    Graph::PlugIn *pEx   = Grid2D::load_graph(cs);
    Graph::PlugIn *pIn   = Amoeba::load_graph(cs);
    Amoeba::PlugIn *pA   = Amoeba::load_connectivity(cs);

    return new Amoeba(*pA, *pP, *pIn, *pEx);
}

BISIM_STARTUP(Amoeba)
BISIM_EXECUTABLE
```

```
// Loads graph from settings.
//
// Get the graph type tag.
// Get the graph identifier.
//
// Arguments variable.
// Assign id to argument.
// Assign overlap setting.
// Assign overwrite setting.
//
// Use args to make graph.
//

// Loads our plug-in:
//
// Arguments variable.
// Assign delay from settings.
// Assign bandwidth setting.
// Use args to make plug-in.
//

// Create our experiment:
//
// Load Grid2D's plug-in.
// Load external graph.
// Load internal graph.
// Load our Amoeba plug-in.
//
// Instantiate our experiment.
//

// Links example code to libraries.
//
```

To compile our program and link in the premade two-dimensional model, make a file, `make_amoeba`, inside `run` and type:

```
g++ -o ../bin/amoeba.exe \
    -L../../../platform/pub/lib \
    -I../../../platform/pub/inc/base \
    ../src/Amoeba.cpp \
    ../src/LossyCAmp.cpp \
    ../src/Proportional.cpp \
    -lmodel-2D \
    -lbisim
```

As usual, to get the program to produce a default settings file for us we can change into the `bin` directory and type: `./amoeba.exe --script-default > settings.txt`. Extra settings have been added by the connectivity model that we have just defined and by the `Grid2D` model, such as the dimensions of a grid and the maximum communication radius of a node. Here are the settings that were produced by the `Amoeba` model. The graph type used in this file is the plug-in called LossyCAmp.

```
[Amoeba] ##################
                          #
bandwidth    333          # Natural - internal amoeba bandwidth (*)
count        10           # Natural - number of amoebae (*)
delay        1            # Natural - internal amoeba transfer delay (*)
graph_id     ioint        #   Label - Unique graph identifier (*)
graph_olap   3            # Natural - Overlap collision (*)
graph_owrite 3            # Natural - Overwrite collision (*)
graph_type   LossyCAmp    #   Label - Effect of signal strength on transfer (*)
machine      amoeba       #   Label - node machine type (*)
                          #
############################
```

Adding the code below to our script file will construct our amoeba virtual machine:

```
[amoeba] ########################
                                 #
$ IO(ioext,0.7)                  #
$ IO(ioint,0.5)                  #
$ Simple(simple)                 #
$ Control(control,5,0.33)        #
$ Assemble(code,program.asm,512) #
                                 #
###################################
```

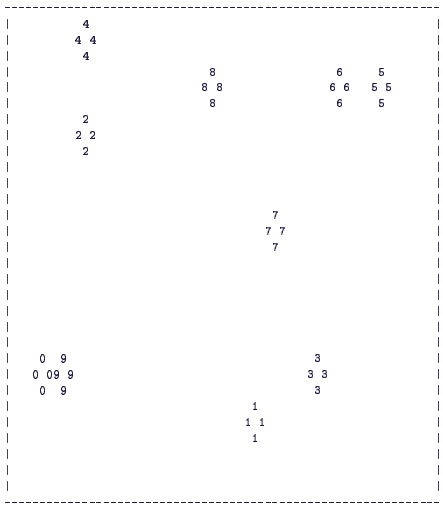
Here, we have two `IO` modules, one with different diffusion coefficients. We include the `Simple` modules for its program control flow abilities. A `Control` module is used with a period of five cycles and diffusion coefficient of a third. Finally, we load our program from a file, `program.asm`, with `512` words of space.

The code below checks whether a node is hungry, and if so, causes it to emit some CAmp. Molecules diffuse through the external and internal parts of the nodes and also pass between them. If the concentration of CAmp is significantly higher in an external part than its corresponding internal part, the cell exerts a force that may cause it to move in that direction.

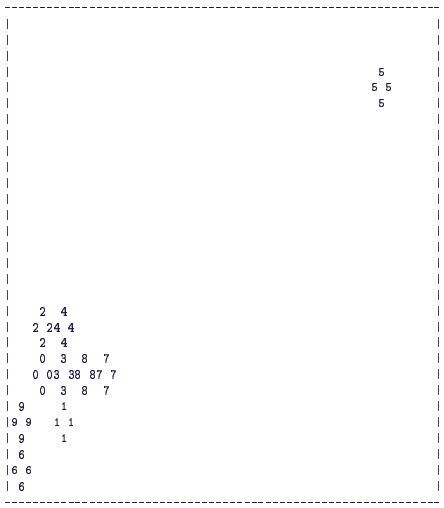
```
.MODEL BISIM_VM                                 ; States the target type as a BiSim VM.

.CODE                                           ; Begins the program code section.

    start:                                      ; Defines our starting point.

    control-food test                           ; Am I hungry?
    simple-bez   test,middle                    ; If not, skip to the middle section.
    ioext-send   1000                           ; Otherwise, diffuse CAmp molecules.

    middle:                                     ; Defines the middle section.

    ioext-fuse   external                       ; Molecules diffuse in from outside.
    ioint-pass   external                       ; Some molecules pass through.
    ioint-fuse   internal                       ; Molecules diffuse within the cell.
    ioext-pass   internal                       ; Some molecules pass through.

    ioint-conc   internal                       ; Check the internal concentration.
    ioext-conc   external                       ; Check the external concentration.

    control-grad test,external,internal         ; Do we have a positive gradient?
    simple-bez   test,start                     ; If not, jump back to the start.
    control-move null                           ; Exert a force on our cell wall.
    simple-jump  start                          ; Repeat back from the start.

.DATA

    null     DW 1
    test     DW 1
    internal DW 1
    external DW 1
```

Now we can play with our experiment. Firstly, try using the provided settings file, `short.txt`, in the `bin` directory before altering it or making up your own. Type: `./amoeba --run-experiment short.txt`. After BiSim's start up information, it displays an simple ASCII grid showing the location of the amoebae made up of four nodes each. The settings file line that toggles whether this is displayed is in the `Grid2D` section and is called `view_debug`.

Start of experiment



A few thousand iterations later



Fiddling with the script file settings, such as the diffusion coefficients can produce different behaviour - from cells wandering *away* from hungry amoebae to amoebae that briefly clump together but bounce off each other.

Well, that ASCII grid is rather primitive, so next we will show how to display the nodes moving and the graph connections forming and breaking while an experiment is running in a graphics window.

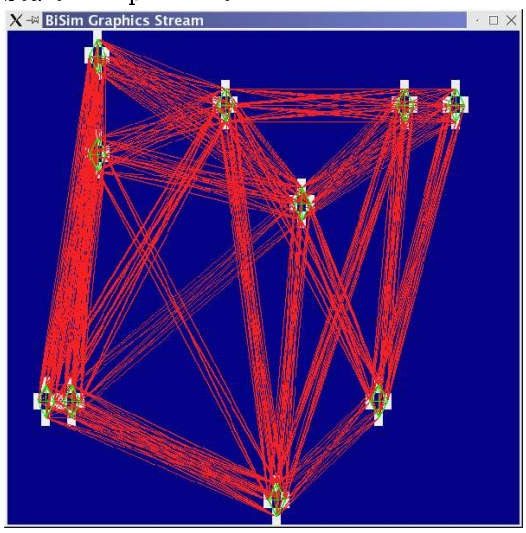
First of all we need to compile the graphics executables. This only needs to be done once. Change into each of the main directories; `display`, `filter` and `viewer` and run the pre-made compilation script by typing `./go`. This will produce the files, `display.exe`, for rendering, `filter.exe`, for redirecting BiSim's output to the rendering engine and two java classes for displaying and refreshing images. Collect these files together and place them either in your path or in the folder in which you run experiments. Inside BiSim's settings file each category that is capable of producing a graphical output stream has an entry called `graphics`. Set each of these to `1` to turn them on and save the modified settings file as `show.txt`.

We can now run and visualise our experiment:

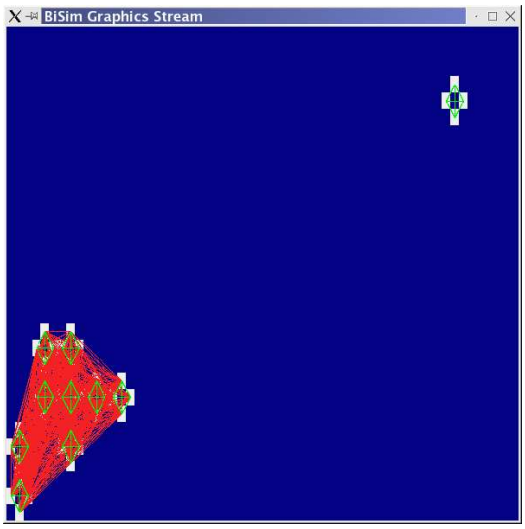
```
java pixserv 3210 sim.pix &
./amoeba.exe --run-experiment show.txt | ./filter.exe \
    | ./display.exe 400 400 java prodport 3210
```

Firstly, an explanation of this long bash command! This runs our `amoeba` experiment with the settings file, `show.txt`. The output of this is piped through a `filter` which passes on any graphics streams to the rendering engine, `display`, which is set to use a resolution of 400x400 pixels. The primitive java `pixserv` program displays an updated image whenever the program `prodport` is called by `display` to notify it that a new image has just been rendered.

Start of experiment



A few thousand iterations later



You can, of course, wrap all of this up in a short batch file and forget about its modularity. Or if you find it useful, you can separately run experiments and collect graphics streams for later viewing. You could also replace `prodport` with a program to enumerate rendered images to produce a movie and so on.

A Programming Interfaces

A.1 Connectivity (Engine)

The base class for connection models is called *Connectivity* and is in the declared in the *engine* directory. It is possible that some methods may not require any code depending on the desired model's purpose.

Public Interface

Connectivity()	Constructs a connection model.
virtual void update(Index ρ_0, const Packets & ρ_1)	A derived class may receive *Packet*s, ρ_1, from *Module* actuators in a *Node* ρ_0, and manipulate the model environment in a user-defined manner.
virtual void initialise(Routes & ρ_0)	This is an opportunity for the connection model to declare which *Graph*s it is using by writing to ρ_0. For example, a model may have two different networks for communication inside a cell and outside a cell. The datatype *Routes* is a map of *Symbol* identifier to *Graph*. This is also a convenient place to manipulate the *Graph* connections before experiments run.
virtual void initialise(Queues &ρ_0)	This call allows a model to pre-load communication signals for *Node*s into ρ_0 before an experiment starts. In detail, a datatype *Queue* is a map of *Symbol* (*Graph* identifier) to a list of *Packets* and the datatype *Queues* is a map of *Graph Index* identifier to a *Queue*.
virtual void stream(Place ρ_0, unsigned ρ_1) const	This method is called for every discrete time point, ρ_1, that passes, including before the experiment begins. ρ_0 is either *start*, *middle*, *end*. The time point is always zero at the *start*, and the maximum number of time steps requested at the *end*, all values in between are labelled as *middle*. Any code for outputting statistics or streaming data over a graphics channel and so on can be placed here.
virtual void refresh()	This method is called at a rate specified in the settings script file. A call to this method may modify the connection *Graph*s. For example, if a *Node* exerts sufficient force in one direction, it may cause it to move, thus changing a connection *Graph*.

Creation

In addition, a model class, *T*, must specify a creation function:

static T *T::create(Base::Setup ρ_0)	This function is called to create the connection model and should return to a *new* connection model object of type *T*. ρ_0 is a channel down which error messages can be sent if the user's code encounters problems.